

PTC UNIT - V

Part-1: Functions: Designing Structured Programs, Functions in C, User Defined Functions, InterFunction Communication, Standard Functions, Passing Array to Functions, Passing Pointers to Functions, Recursion

Part-2: Files I/O - Text Input / Output: Files, Streams, Standard Library Input / Output Functions, Formatting Input / Output Functions, Character Input / Output Functions

Files I/O - Binary Input / Output: Text versus Binary Streams, Standard Library, Functions for Files, Converting File Type.

Functions in C

Designing Structured Programs in C

Structured programming is a **modularization** or a **parsing** technique that divides a larger program into smaller subprograms. The 3 methods of parsing or dividing are

1. **Top-down:**

Breaks down a large problem into smaller “tasks” namely functions and sub-functions.

2. **Bottom-up:**

Starts with basic building blocks (like Lego) and builds smaller code units. These lower functions work together and emerge to solve the bigger problem.

3. **Middle-out:**

The best strategy is to design a program by combining and using both methods at different levels of parsing.

Designing Structured programming makes a program

- easy to understand,
- easy to implement,
- easy to test and debug,
- easy to improve quality, and
- code reusable.

Structured programming enables code reusability, which means writing code once and using it many times.

In C, **Top-Down structured programming** can be designed **using the concept of the functions**. Using the concept of the functions, we can divide the larger program into smaller function modules. These functions are executed individually.

PTC UNIT - V

A C program is written with one or more functions. The `main()` function is mandatory to execute a C program. The execution of the program always starts with `main()`, but it can also **call other functions to perform specific tasks**.

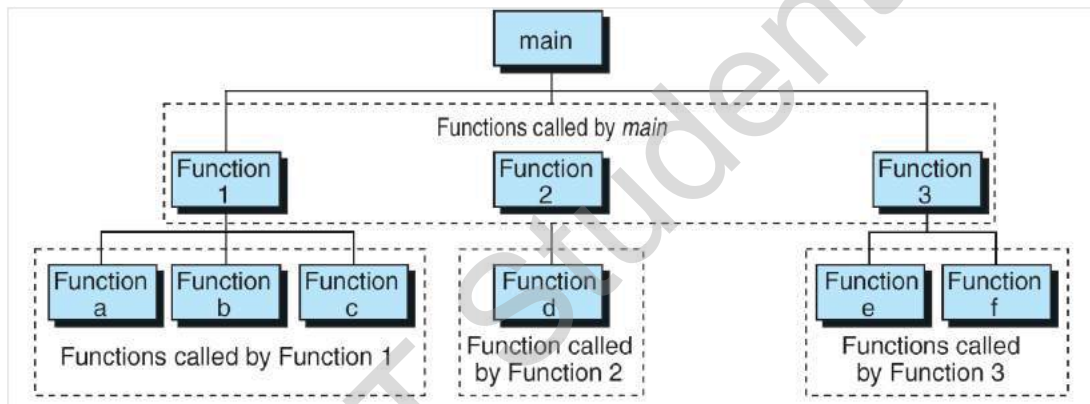
Definition of a Function:

A function is a set of statements used to perform a specific task and is executed individually. A function is a small part of a program.

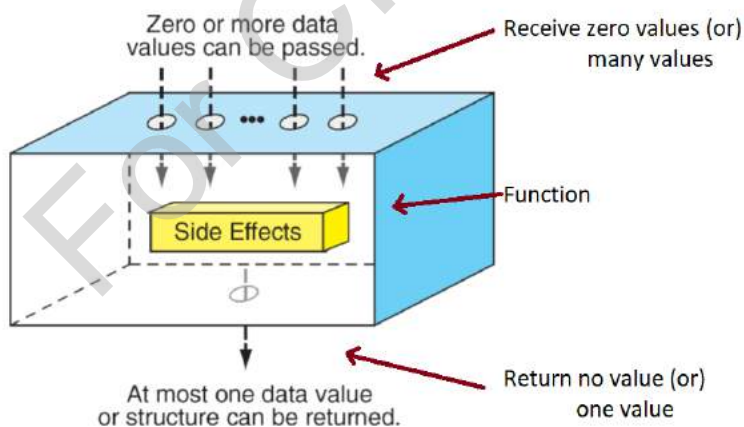
The **purpose** of a function is

- to receive zero or more data values,
- operate on those data values, and
- return at most one data value as a result.

Chart using Functions in C:



Function Concept:



Note: A function can have a return value, a side effect, or both. The side effect occurs before the value is returned from the function. (Ex: `return (x=y++);` //x value is returned, but y value also changed as a side effect).

PTC UNIT - V

The functions are of 2 types.

Types of Functions	
A. User-Defined Functions	B. Standard C Library Functions

A. User-Defined Functions:

Users can define their own functions in a C program. The functions that are created by users are called user-defined functions.

- A function is a block of code used to perform a specific task.
- A function runs only when it is called.
- We can pass data into functions as parameters.
- Functions can be reused; write the code once, and use it many times.

The user-defined functions in C consist of the following:

1. **Function Declaration (or Function Prototype)** - tells the compiler about a function's name, return type, and parameters.
2. **Function Definition** - provides the actual body of the function.
3. **Function Call** - can be done from main(), another function, or by itself to execute the called function.
4. **A Function in C must be Declared and Defined before the Function Call.**

Optimized Layout of C Program using Functions: A function declaration should be before the main() and a function Definition should be after main() for an optimized C program. (A function may also be declared inside the main function or in any other function which is not optimized).

<pre>#include<stdio.h> Function Declarations</pre>
<pre>int main() { Local declaratons Function calls Statements }</pre>
<pre>Function Definitions { Local declarations Statements }</pre>

1. Function Declaration

The **function declaration** is also called a **function PROTOTYPE**. A function prototype consists of

- **name of the function**
- **data type of the return value and**
- **list of parameters.**

Declaration Syntax:

```
returnType functionName(parameters list);
```

returnType specifies the data type of the value which is sent out as a return value from the function definition.

functionName is a user-defined name used to identify the function uniquely in the program.

parametersList is the data values that are sent to the function definition.

Ex: **int multiply(int x, int y);**

2. Function Definition

The function definition provides the actual code of that function. The function definition is also known as the **body of the function**. The function definition means the actual statements to be executed by a function. These statements of a function are written inside the curly braces "{ }". The function definition is executed only when it is called from other functions or by itself.

Function Definition Syntax:

```
return_type function_name( parameter list )
{
    body of the function
}
```

Ex: **function definition**

```
int multiply(int x, int y)
{ int result;
  result = x * y;
  return result;
}
```

3. Function Call

When a program calls a function, the program control is transferred to the called function. A called function performs a defined task and when its return statement is executed or when its function-ending closing brace is reached, it returns the program control back to the main program.

A function is called either from the main function or from other functions or also by itself.

Function Call Syntax:

```
functionName(parameters);
```

Example: Program to multiply two integers using functions.

```
#include <stdio.h>
/* function declaration */
int multiply(int x, int y);

int main () {
    /* local variable definition */
    int a = 10;
    int b = 20;
    int result;

    /* calling a function to get multiplication value */
    result = multiply(a, b);

    printf( "Multiplied value is : %d\n", result);

    return 0;
}

/* function returning the multiplication of two numbers */
int multiply(int x, int y)
{
    /* local variable declaration */
    int mult;
    mult = x * y;
    return mult;
}
```

Output: Multiplied value is : 200

Parameter Passing in Functions

When the execution control is transferred from **calling function** to **called-function**, it may carry none or many data values. The data that are passed from the calling function to called function are called **actual parameters**.

Parameters are the data values that are passed from the calling function to called function.
 * At a function call, the copy of actual parameter values are copied into formal parameters.

In C, there are two types of parameters.

- **Actual Parameters (or Arguments)**
- **Formal Parameters**

The **actual parameters** are the parameters that are specified in the **calling function**.

The **formal parameters** are the parameters that are declared in the **called function**.

When a function is called, a COPY of Actual parameter values is copied into Formal parameters.

Actual Parameters vs Formal Parameters:

Actual Parameters (or Arguments)	Formal Parameters
Actual parameters are passed from the CALLING function. These are also called Arguments .	Formal parameters are declared in the CALLED function definition. These are also called Parameters .
Actual parameters are SENDING parameters to called (receiving) function from the calling (sending) function. Ex: in main()	Formal parameters are declared in the function definition for RECEIVING values from Actual Parameters..
During the time of call, each argument is always assigned to the parameter in the function definition.	Parameters are local variables that are assigned values of the arguments when the function is called.
Must match <ul style="list-style-type: none"> ● exact data type ● number of parameters ● order of parameters 	Must match <ul style="list-style-type: none"> ● exact data type ● number of parameters ● order of parameters
You may have the same or different names for Formal and Actual parameters. Ex: <code>int q; float p; cost(q, p); //function call</code>	You may have the same or different names for Formal and Actual parameters. Ex: <code>float cost(int qty, float price);</code>

Example: Program to add two integers using a function

```

#include<stdio.h>
int add(int a, int b); //Declaration of Function

int main()
{int x=10, y = 20, total;
  total = add(x,y); //x and y are ACTUAL parameters
  printf("Total of x and Y = %d", total);
  return 0;
}
//Definition of Function
int add(int a,int b) //a and b are FORMAL parameters
{ int sum;
  sum = a + b;
  return sum; //Sends value back to source
}

```

In the concept of functions, the function call is known as "**Calling Function**" and the function definition is known as "**Called Function**".

When we make a function call, the execution control jumps from the calling function to called function. After executing the called function, the execution control comes back to the calling function from the called function. When the control jumps from calling function to called function it may carry one or more data values called "**Parameters**" and while coming back it may carry a single value called "**return value**". That means the data values transferred from the calling function to called function are called **Parameters** and the data value transferred from called function to the calling function is called **Return value**.

Based on the data flow between the calling function and called function, the functions are classified into 4 categories as described below.

Advantages Of Functions

- We can implement modular programming using functions
- Larger programs can be divided into smaller modules using functions
- Functions make the program easy to read and understand
- Once a function is created it can be used many times (code re-usability)

Inter-Function Communication

The exchange of information using arguments between two functions is called an Inter-Function Communication.

A Function Call in the program transfers the execution control from Calling a Function to Called Function and executes the function definition. After the function is executed, the control finally comes back to the Calling Function.

In this process, both Calling and Called functions exchange information. The process of exchanging information between Calling and Called functions is called Inter-Function Communication.

What are the Categories of Functions?

The Inter-Function Communication in C is categorized into 4 categories as follows. The 4 categories of functions are divided based on **Parameter Passing** and **Return value**.

1. **Bi-directional Communication**
2. **Downward Communication**
3. **Upward Communication**
4. **No Exchange Communication**

Category	Pass Parameters	Return Value	Example
1. Bi-Directional	Yes	Yes	<code>int add(int, int)</code>
2. Downward	Yes	No	<code>void add(int, int)</code>
3. Upward	No	Yes	<code>int add()</code>
4. No Transfer	No	No	<code>void add()</code>

1. Bi-Directional Function: Pass Parameters - YES & Return Value - YES

When these types of functions are used,

- Calling-functions send parameters to called function, and
- Called function returns value back to the calling function.
- Execution control jumps from the calling function to called function along with parameters and executes called function, and
- Finally comes back to the calling function along with a return value.

Example:

```
// Function category: Passing arguments into function & Returns value
from function
#include <stdio.h>
int add(int, int);
void main() {
    int a=10, b=20;
    printf("Addition of 2 numbers = %d\n", add(a, b));
}
int add(int x, int y)
{
    return (x+y);
}
```

Output: Addition of 2 numbers = 30

2. Downward: Pass Parameters - YES & Return Value - NO

When these type of functions are used,

- Calling-functions send parameters to called-function and
- Called function DOES NOT return value back to calling-function.
- Execution control jumps from calling function to called function along with parameters and executes called function, and
- Finally comes back to the calling function WITHOUT a return value.

Example:

```
// Function category: Passing arguments into function & NO Return
value from function
#include <stdio.h>
void add(int, int); //Function Declaration
void main() {
    int a=10, b=20;
    add(a, b); //Function Call
}
```

```
void add(int x, int y)    //Function Definition
{ printf("Addition of 2 numbers = %d\n", x+y);
  //NO return value from function
}
```

Output: Addition of 2 numbers = 30

3. Upward: Pass Parameters - NO & Return Value - YES

When these type of functions are used,

- Calling-functions DOES NOT send parameters to called-function and
- But the Called function returns value back to calling-function.
- Execution control jumps from calling function to called function with NO parameters and executes called function, and
- Finally comes back to the calling function with a return value.

Example:

```
// Function category: Passing NO arguments into function & YES Returns
value from function
#include <stdio.h>
int add(); //Function Declaration
void main()
{ int result;
  result = add(); //Function Call
  printf("Addition of 2 numbers = %d\n", result);
}
int add() //Function Definition
{ int x=10, y=20;
  return (x+y);
}
```

Output: Addition of 2 numbers = 30

4. No Transfer: Parameters - NO & Return Value - NO

When these type of functions are used,

- Calling-functions DOES NOT send parameters to called-function and
- Called function also DOES NOT return value back to calling-function.
- Execution control jumps from calling function to called function with NO parameters and executes called function, and
- Finally comes back to the calling function with NO return value.

PTC UNIT - V

Example:

// Function category: Passing NO arguments into function & NO Return value from function

```
#include <stdio.h>
void add(); //Function Declaration
void main() {
    add(); //Function Call
}
void add() //Function Definition
{
    int x=10, y=20;
    printf("Addition of 2 numbers = %d\n", x+y);
}
```

Output: Addition of 2 numbers = 30

B. Standard C Library Functions

The library of C Programming Language comes with many pre-defined functions. These functions make programming easier. These pre-defined functions are known as Library Functions or Standard Functions or System Defined functions.

The functions that are defined within the C library are called standard or system-defined functions.

All the standard functions are defined inside the header files such as `stdio.h`, `conio.h`, `math.h`, `string.h`, etc. Ex: The functions **printf()** and **scanf()** are defined in the header file called **stdio.h**.

Whenever we use standard functions in the program, we must include the respective header file using `#include` statement. Ex: If we use a standard function **sqrt()** in the program, we must include the header file called **math.h** because the function **sqrt()** is defined in **math.h**.

Header file	Description	Standard C Functions (few)
stdio.h	This is standard input/output header file	scanf() , printf() , fscanf() , fprintf() , fopen() , fclose() , fread() , fwrite() , fseek() , ftell() , rewind() , remove() , rename() , feof() , ferror() , clearerr() , freopen()
conio.h	This is console input/output header file	clrscr() , getch()
string.h	All string related functions are defined in this header file	strlen() , strcpy() , strupr() , strlwr()
stdlib.h	This header file contains general functions used in C programs	atoi(s) , atol(s) , atof(s) • converts character to float, integer and long respectively
math.h	All maths-related functions are defined in this header file	sqrt() , abs() , sin() , cos() , exp() , log() , pow() , ceil() , floor()
time.h	This header file contains time and clock related functions	clock() , time() , localtime() ,
ctype.h	All character handling functions are defined in this header file	isalnum(c) , isalpha(c) , isdigit(c) , isctrl(c) , ispunct(c) punctuation, isspace(c) space, tab or new line isupper(c) 'A'-'Z'
stdarg.h	Variable argument functions are declared in this header file	va_start() , va_arg() and va_end()
locale.h	Sets or reads location dependent information	setlocale() , localeconv()

PTC UNIT - V

errno.h	Error handling functions are given in this file	extern int errno
graphics.h	Provides functions to draw graphics.	circle(), rectangle()
float.h	Provides constants related to floating point data values	
stddef.h	Defines various variable types	

Example: Program using standard C mathematical functions

```
#include<stdio.h>
#include <math.h>
#define PI 3.1415
int main(){
printf("\nCeiling = %f",ceil(3.6));
printf("\nFloor = %f",floor(3.2));
printf("\nSquare root = %f",sqrt(7));
printf("\nPower = %f",pow(3,3));
printf("\nAbsolute = %d",abs(-12));
printf("\nRemainder = %.0f",fmod(7,2));

double x, fractpart, intpart;
x = 7.54321;
//returns the fraction part and sets integer to the integer part
fractpart = modf(x, &intpart);
printf("\nInteger part = %lf", intpart);
printf("\nFraction Part = %lf", fractpart);

double d, deg, val;
d = 60.0;
val = PI / 180.0;
deg = cos( d*val );
printf("\nThe cosine of %lf is %lf degrees", d, deg);
return 0; }
```

Output:

```
Ceiling = 4.000000
Floor = 3.000000
Square root = 2.645751
Power = 27.000000
```

```
Absolute = 12
Remainder = 1
Integer part = 7.000000
Fraction Part = 0.543210
The cosine of 60.000000 is 0.500027 degrees
```

Methods of Function Call:

There are two methods to call a function by passing parameters from calling function to called function.

1. Call by Value
2. Call by Reference (also called Passing Pointer to Function)

1. Call by Value

In this method, the actual parameter values are COPIED to formal parameters. These duplicated formal parameters are used in the called function. The changes made to the formal parameters do not affect the values of the actual parameters.

That means, after the execution control comes back to the calling function, the actual parameter values remain the same.

Example: Program for Function Call by Value - Swap Numbers

```
#include <stdio.h>
void swap( int p, int q);
int main( )
{
int a=35, b=45 ;
printf("Before swapping: %d, %d", a, b);

swap(a, b); //calling function BY VALUE or copies of values
printf("\nAfter swapping: %d, %d", a, b);
}
void swap( int p, int q) //called function
{ int tmp;
  tmp = p;
  p = q;
  q = tmp;
}
```

Output:

Before swapping: 35, 45

After swapping: **35, 45**

In the above example program, the variables a and b are called actual parameters and the variables p and q are called formal parameters. The value of a is copied into p and the value of b is copied into q. The changes made on variables p and q do not affect the values of a and b.

2. Call by Reference

In the call by reference method, the **memory addresses of the actual parameters are passed to called function and received by the formal parameters (pointers)**. This address is used to access the memory locations of the actual parameters in the called function. Here, the formal parameters must be pointer variables.

Whenever we use these formal parameters in the called function, they directly access the memory locations of actual parameters. So the changes made to the formal parameters affect the values of actual parameters.

Example: Program for Function Call by Reference - Swap Numbers

```
#include <stdio.h>
void swap( int *p, int *q);

int main( )
{
    int a=35, b=45 ;
    printf("Before swapping: %d, %d", a, b);

    swap(&a, &b);    //calling function BY REFERENCE or sending addresses
    printf("\nAfter swapping: %d, %d", a, b);
}

void swap( int *p, int *q)    //called function
{
    int tmp;
    tmp = *p;
    *p = *q;
    *q = tmp;
}
```

Output:

Before swapping: 35, 45

After swapping: **45, 35**

PTC UNIT - V

In the above example program, the addresses of variables **a** and **b** are copied to pointer variables **p** and **q**. The changes made on the pointer variables **p** and **q** in called function affect the values of actual parameters **a** and **b** in the calling function.

Summary of Actual Parameters (Arguments) vs Formal Parameters:

Function Call by Value

- Actual Parameters are Copied to the Formal Parameters
- Changes to Formal Parameters in function don't affect Actual Parameters.

Function Call by Reference

- Address of Actual Parameters is Passed to the Formal Parameters (POINTERS)
- Changes to Formal Parameters affect the value of Actual Parameters.

Scope of C Variable:

A scope is an area of a program where a variable can be accessed after its declaration.

Every variable in C is defined in scope. Scope is the section or region of a program where a variable has its life, and that variable cannot be used or accessed outside that region.

A variable declared within a function is different from the variable declared outside of a function. The variable can be declared in 3 places. They are

Type	Position	Access
1. Global Scope Variables	Before main() function. Out of all functions. Accessible in all functions.	Accessible to all functions.
2. Local Scope Variables <ul style="list-style-type: none">• Local to a Function• Local to a Block { }	Declaration of variables inside a function or inside a block { }.	Accessible within a function or a block { }.
3. Local Scope Formal Parameters <ul style="list-style-type: none">• Local to a Function	In the header of the function definition as formal parameters.	Accessible within a function.

1. Global Scope (Global Declaration Before the function definition)

Variables declared outside the function block and accessed inside the function are called global variables.

Global Scope

- The global variables are defined outside a function or any specific block on top of the C program. These variables hold their values throughout the program
- These are accessible in all functions defined in your program.
- Any function can access variables defined within the global scope, i.e., its availability stays for the entire program after being declared.

2. Local Variables

The variables that are declared within a function block are used only within that function. Those variables are not accessible outside that function. These are called local variables.

Local Function Scope or Block Scope

- A local scope or block is a collective program statement placed and declared within a function or block within curly braces.
- Variables lying inside such blocks are called local variables to those blocks. All these locally scoped statements are written and enclosed within the curly braces { }.
- C also has a provision for nested blocks, which means that a block or function can occur within another block or function. That means, the variables declared within a parent block can be accessed in the nested block and all other internal blocks. However, these variables cannot be accessed outside the parent block.

Example: Program to show Global and Local Scope Variables

```
#include<stdio.h>
int num1, num2; //Declaration of Global Variable
void add();
void sub();
void mult();
int main(){
    num1 = 10 ; //Initialization of Global Variable
    num2 = 20 ; //Initialization of Global Variable
    printf("num1 = %d, num2 = %d \n", num1, num2) ;
    add();
    sub();
    mult();
    return 0; }
```

```

void add()
{
    int result ; //Local to add()
    result = num1 + num2 ; //accessing Global variables
    printf("\nAddition = %d", result) ;
}
void sub()
{
    int result ; //Local to sub()
    result = num1 - num2 ; //accessing Global variables
    printf("\nSubtraction = %d", result) ;
}
void mult()
{
    int result ; //Local to mult()
    result = num1 * num2 ; //accessing Global variables
    printf("\nMultiplication = %d", result) ;
}

```

Output:

```

num1 = 10, num2 = 20
Addition = 30
Subtraction = -10
Multiplication = 200

```

3. Local Scope Formal Parameters in Function Definition:

The variables declared in the function definition as parameters have a local variable scope. These variables behave like local variables in the function. They can be accessed inside the function; they are not accessible outside the function.

Example: Program to show Local Scope Formal Parameter in Function

```

#include<stdio.h>
void addition(int, int) ;

int main(){
    int num1, num2 ;
    num1 = 10 ;
    num2 = 20 ;
    addition(num1, num2) ;
    return 0;    }

```

```

void addition(int a, int b)
{ //a and b are Local Formal Parameters
  int resultSum ;
  resultSum = a + b ;
  printf("\nAddition = %d", resultSum) ;
}

```

Output: Addition = 30

Note: Here, the variables **a** and **b** are declared in function definition as parameters. So, they can be used only inside the addition() function.

Passing Arrays to Function:

An array is a collection of similar data types which are stored in memory as a contiguous memory block.

We can pass the address of an array to a function argument. For example, we have a function to sort a list of numbers; it is more efficient to pass these numbers as an array to function instead of passing them as variables. Since the number of elements the user has is not fixed and passing numbers as an array will allow our function to work for any number of values.

Since arrays are a continuous block of values, we can pass the reference of the first memory block of our array to the formal parameter in the function. This receiving parameter can be an array variable or can also be a pointer variable. And then we can easily calculate the address of any element in the array using the formula:

address(arr[i]) = (start address of array) + i * (size of individual element)

(see example-2)

Note: Name of the array acts as a pointer because it points to address of the first element of the array. Ex: int arr[10]; here, the name “**arr**” and “**&arr[0]**” both refer to the address of the first value in the array. So, we can send an Actual parameter such as “**arr**” to a formal parameter of either “**a[]**” or “***a**” as both will receive the address.

Example 1 of 2: Program to Pass an Array to Function and Access the Array

```

//Pass arrays to function
#include<stdio.h>
void read(int[], int); //No need to specify parameter names in Declaration
void display(int[], int); //No need to specify parameter names in Declaration
void main() {
  int qty[10], n, j;
  printf("Enter number of products : ");
}

```

PTC UNIT - V

```
scanf("%d", &n);
read(qty, n);
display(qty, n);
}
void read(int r[10], int n) {
    int i;
    printf("\nEnter Quantity Sold for %d Products : \n", n);
    for(i = 0; i < n; i++) {
        printf("Enter Quantity Sold for Product x[%d] : ", i);
        scanf("%d", &r[i]);
    }
}
void display(int d[10], int n) {
    int i;
    printf("\nThe Sold Quantity are : \n");
    for(i = 0; i < n; i++) {
        printf("Product d[%d] : %d\n", i, d[i]);
    }
}
```

Output:

Enter number of products : 3

Enter Quantity Sold for 3 Products :

Enter Quantity Sold for Product x[0] : 100

Enter Quantity Sold for Product x[1] : 200

Enter Quantity Sold for Product x[2] : 300

The Sold Quantity are :

Product y[0] : 100

Product y[1] : 200

Product y[2] : 300

Example 2 of 2: Program to Pass an Array to Function and Access with Pointer Arithmetic

```
//Pass arrays to function parameter pointer (with pointer arithmetic)
#include<stdio.h>
void readScore(int *s, int size);
void displayScore(int *s, int size);
int totalScore(int *s, int size);
int main()
{
    int n = 11; //max number of players
    int scores[n], total;
```

PTC UNIT - V

```
    readScore(scores, n); // "scores" is starting address of the array
    displayScore(scores, n);
    total = totalScore(scores, n);
    printf("\n Total = %d", total);
return 0;
}
void readScore(int *s, int size)
{
    int i;
    printf("Enter 11 scores: \n");
    for(i = 0; i < size; i++) {
        printf("Player %d = ", i+1);
        scanf("%d", (s + i));
    }
}
void displayScore(int *s, int size)
{
    int i;
    for(i = 0; i < size; i++)
    {
        printf("\nPlayer %d = %d ", i+1, *(s + i));
    }
}

int totalScore(int *s, int size)
{
    int i, sum=0;
    for(i = 0; i < size; i++)
    {
        sum += *(s + i);
    }
return sum;
}
```

Output:

```
Player 1 = 10
Player 2 = 40
Player 3 = 50
Player 4 = 10
Player 5 = 30
Player 6 = 60
Player 7 = 25
Player 8 = 25
Player 9 = 25
Player 10 = 25
Player 11 = 50
```

Total = 350

Passing Pointers to Function:

Passing the pointers to the function means the memory location of the variables is passed to the parameters in the function, and then the operations are performed. The function definition accepts these addresses using pointers.

Example: Program to Pass Pointer to a Function (Or You may also use the swap() program given above under the “call by reference” section as that too will pass a pointer to the function).

```
#include <stdio.h>
void increment(int* ptr) {
    (*ptr)++; // adding 1 to *ptr variable value
}
void decrement(int* ptr) {
    (*ptr)--; // subtracting 1 from *ptr variable value
}

int main()
{
    int* p, i = 10;
    p = &i;
    increment(p);
    printf("%d", *p); // prints 11

    decrement(p);
    printf("%d", *p); // prints 10
    return 0;
}
```

Output: 11 10

Recursive Functions in C

In C, function calls can be started,

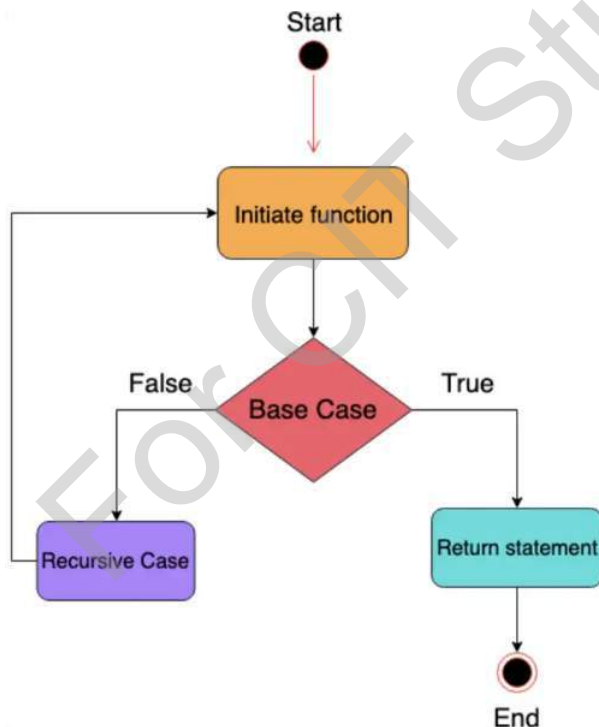
- from the main() function,
- from another function, or
- **from the same function itself.**

Definition of Recursive Function:

A function called by itself repetitively is called a recursive function. The function call is termed a recursive call.

The recursive function will call itself multiple times until a condition is satisfied. The recursive functions should be used very carefully because, when a function is called by itself it enters into the infinite loop. And when a function enters into the infinite loop, the function execution never gets completed. We should define the condition to exit from the function call so that the recursive function gets terminated.

Flowchart of Recursive Function:



The recursive function has two main parts in its body,

1. **the base case** (condition) and
2. **the recursive case** (recursive function call).

The flow of execution of Recursive Function:

- first, the program checks the base case condition.
- If it is TRUE, the function returns and quits;
- otherwise, the recursive case is executed by calling the function recursively.

The representation of recursion in a C program is as follows.

```
recursive_function()
{
    //base case condition
    if base_case = true;
        return;
    else
        //recursive case
        return recursive_function(argument); // recursive call
}
```

Note: When a function is called by itself, the first call remains under execution till the last call is invoked. Every time when a function call is invoked, the function returns the execution control to the previous function call.

Types of Recursion in C

There are two types of recursion in the C language.

1. Direct Recursion
2. Indirect Recursion

1. Direct Recursion

Direct recursion in C occurs when a function calls itself directly from inside. Such functions are also called direct recursive functions.

Structure of direct recursion:

```
function1 ()
{
    //some code
    function1 ();
    //some code
}
```


In the direct recursion structure, the `function1 ()` executes, and from inside, it calls itself recursively.

Example 1 of 2: Program to find Factorial of an integer using Direct Recursion Function

```
#include<conio.h>
int factorial( int ) ;
int main()
{
    int fact, n ;
    printf("Enter a positive integer: ") ;
    scanf("%d", &n) ;
    fact = factorial( n ) ;
    printf("\nFactorial of %d is %d\n", n, fact) ;
    return 0;
}
int factorial(int n)
{
    int temp ;
    if(n <= 1)
        return 1 ;
    else
        temp = n * factorial( n-1 ) ; // recursive function call
    return temp ;
}
```

Output:

```
Enter a positive integer: 3
Factorial of 3 is 6
```

Explanation:

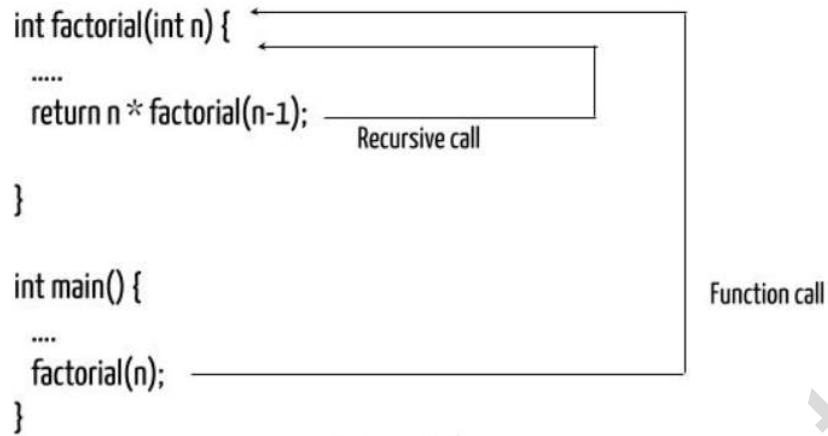
In the above example program, the `factorial()` function call is initiated from `main()` function with the value 3. Inside the `factorial()` function, the function calls `factorial(2)`, `factorial(1)`, and `factorial(0)` are called recursively.

In this program execution process,

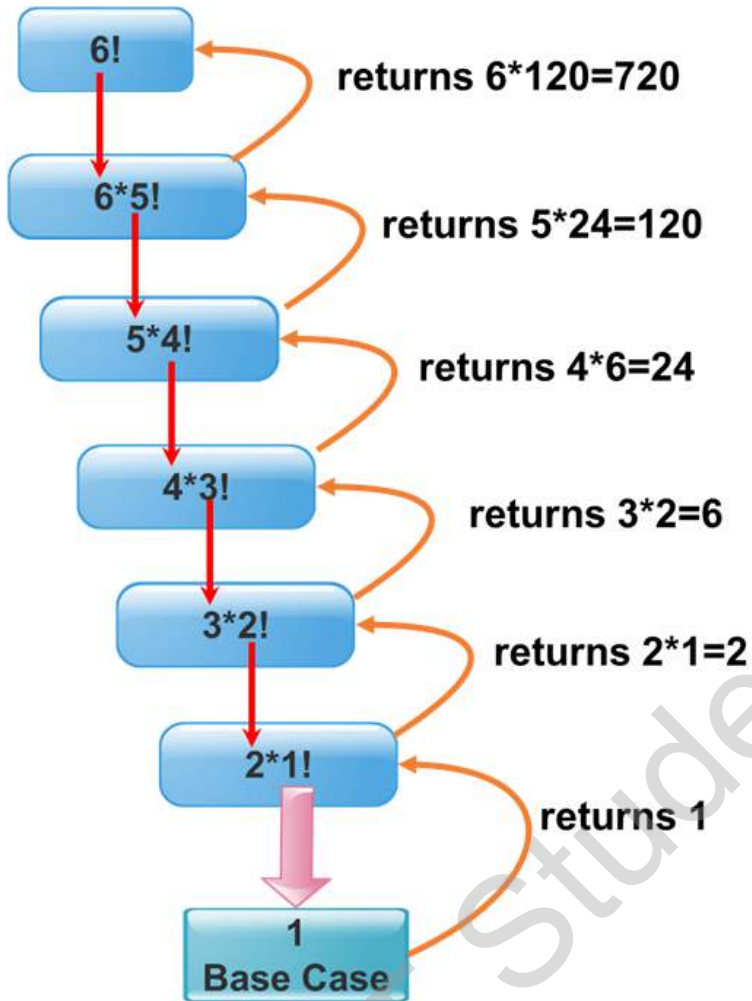
- the function call `factorial(3)` remains under execution till the execution of function calls `factorial(2)`, `factorial(1)`, and `factorial(0)` gets completed.
- Similarly the function call `factorial(2)` remains under execution till the execution of function calls `factorial(1)` and `factorial(0)` gets completed.
- In the same way the function call `factorial(1)` remains under execution till the execution of function call `factorial(0)` gets completed.

PTC UNIT - V

The following figure shows the complete execution process of the above program.



Ex: The recursive function call execution to find factorial of 6. The function calls are stored in the stack until the control reaches the base case condition; then the calculation will occur. **The execution will occur in reverse order; the Last call is executed 1st then 2nd, & 3rd so on**



Ex: The recursive function call execution to find factorial of 3.

PTC UNIT - V

```
int factorial(int);  
void main()  
{  
    int fact, n;  
    printf("Enter any positive integer: ");  
    scanf("%d", &n);  
    fact = factorial(n);  
    printf("Factorial of %d is %d", n, fact);  
}
```

factorial(3);

6

```
int factorial(int n)  
{  
    int temp;  
    if(n == 0)  
        return 1;  
    else  
        temp = n * factorial(n-1);  
    return temp;  
}
```

3*factorial(2);

2

```
int factorial(int n)  
{  
    int temp;  
    if(n == 0)  
        return 1;  
    else  
        temp = n * factorial(n-1);  
    return temp;  
}
```

2*factorial(1);

1

```
int factorial(int n)  
{  
    int temp;  
    if(n == 0)  
        return 1;  
    else  
        temp = n * factorial(n-1);  
    return temp;  
}
```

1*factorial(0);

1

```
int factorial(int n)  
{  
    int temp;  
    if(n == 0)  
        return 1;  
    else  
        temp = n * factorial(n-1);  
    return temp;  
}
```

Example 2 of 2: Program to find Fibonacci Series of a given number of terms using Direct Recursion Function

```

#include<stdio.h>
int fibonacci(int i);
int main()
{
    int i, n;
    printf("Enter terms for fibonacci series: ");
    scanf("%d", & n);

    for (i = 0; i < n; i++) {
        printf(" %d ", fibonacci(i));
    }
    return 0;
}

int fibonacci(int i)
{
    if (i == 0) {
        return 0;
    }
    if (i == 1) {
        return 1;
    }
    return fibonacci(i - 1) + fibonacci(i - 2);
}

```

Explanation:

In the C program above, we have declared a function named fibonacci(). It takes an integer i as input and returns the ith element of the Fibonacci series. At first, the main() function will be executed where we have taken two variables i and n. We will take input from the user that will be stored in n, and the for loop will execute till n iteration where with each iteration, it will pass the parameter to fibonacci() function where the logic for the Fibonacci series is written. Now inside fibonacci() function, we have nested if-else. If input = 0, it will return 0, and if the input = 1, it will return 1. These are the base cases for the Fibonacci function. If the value of i is greater than 1, then fibonacci(i) will return fibonacci(i - 1) + fibonacci(i - 2) recursively, and this recursion will be computed till the base condition.

2. Indirect Recursion

Indirect recursion in C occurs when the first function calls the second function and the second function calls the first function again. Such functions are also called indirect recursive functions.

Following is the structure of indirect recursion.

```
function1 ()
{
    //some code
    function2 ();
}
```

```
function2 ()
{
    //some code
    function1 ();
}
```

Example: Program with Indirect Recursion Function

This program prints numbers from 1 to 10 in such a manner that when an odd no is encountered, we will print that number plus 1. When an even number is encountered, we would print that number minus 1 and will increment the current number at every step.

```
//Prints 1-10; odd+1 and even-1 using Indirect Recursion Function
#include<stdio.h>
void odd();
void even();
int n=1;

void odd()
{
    if(n <= 10)
    {
        printf("%d ", n+1);
        n++;
        even();
    }
    return;
}

void even()
{
    if(n <= 10)
    {
        printf("%d ", n-1);
        n++;
        odd();
    }
    return;
}
```

```
}  
  
int main()  
{  
    odd();  
}
```

Output:

2 1 4 3 6 5 8 7 10 9

Advantages of Recursion:

1. The code becomes shorter and reduces the unnecessary calling to functions.
2. Useful for solving formula-based problems and complex algorithms.
3. Useful in Graph and Tree traversal as they are inherently recursive.
4. Recursion helps to divide the problem into sub-problems and then solve them, essentially divide and conquer.

Disadvantages of Recursion:

1. The code becomes hard to understand and analyze.
2. A lot of memory is used to hold the copies of recursive functions in the memory.
3. Time and Space complexity is increased.
4. Recursion is generally slower than iteration.

Summary of Recursion:

- There are two types of recursion in the C language. The first is Direct recursion and Indirect recursion.
 - The Direct recursion in C occurs when a function calls itself directly from inside.
 - Indirect recursion occurs when a function calls another function, and then that function calls the first function again.
 - The function call to itself is a recursive call, and the function will become a recursive function.
 - The stack is maintained in the memory to store the recursive calls and all the variables with the value passed in them.
-

PTC UNIT - V Part-2 - File I/O

Part-1: Functions: Designing Structured Programs, Functions in C, User Defined Functions, InterFunction Communication, Standard Functions, Passing Array to Functions, Passing Pointers to Functions, Recursion

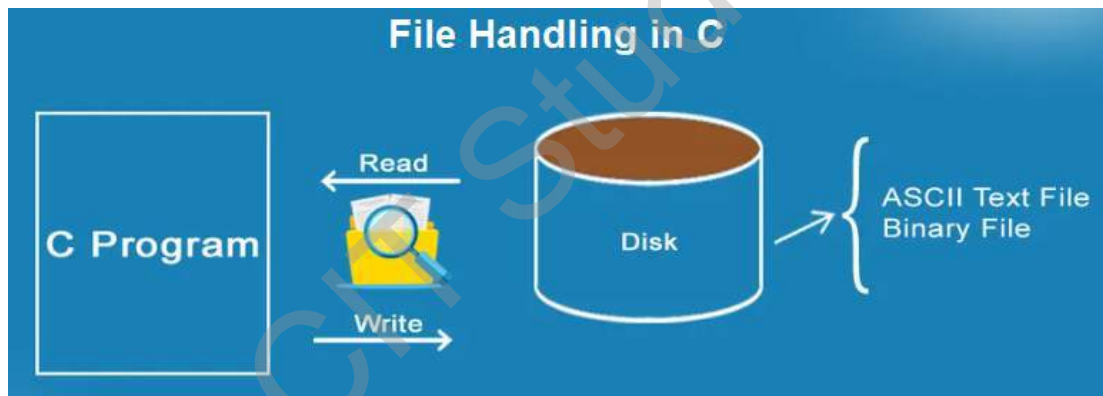
Part-2: Files I/O - Text Input / Output: Files, Streams, Standard Library Input / Output Functions, Formatting Input / Output Functions, Character Input / Output Functions

Files I/O - Binary Input / Output: Text versus Binary Streams, Standard Library, Functions for Files, Converting File Type.

File Handling in C

A file is a place where a program stores data permanently in a sequence of bytes.

The content available on a file is not volatile like the compiler memory in C. But the program can perform various file operations, such as creating, opening, reading a file, or manipulating the data present inside the file. This process is known as file handling in C.



Need for file handling in C - The output generated by a program needs to be saved permanently to check or use at various times.

What are the types of file streams in C?

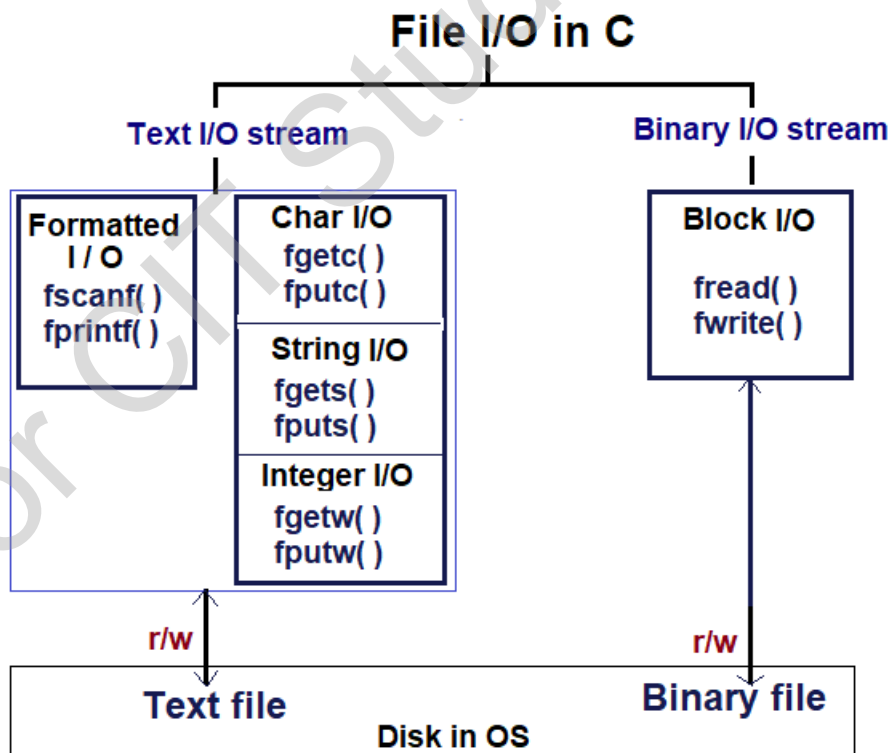
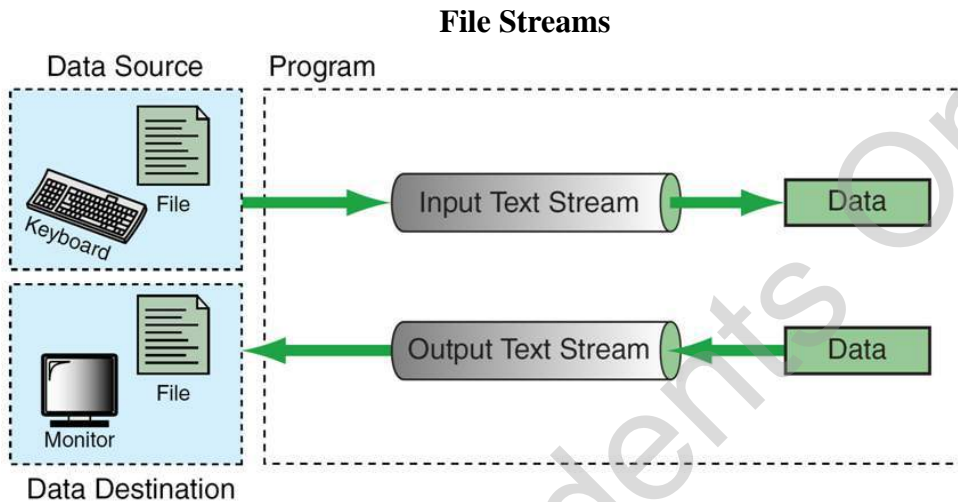
We have basically 2 types of file streams in C. They are namely Text I/O stream and Binary I/O stream.

1. **TEXT I/O stream** - The user can create TEXT Files easily using file handling in C. It stores information in the form of ASCII characters internally and when the file is opened, the content is readable by humans. It can be created by any text editor with a **.txt** or **.rtf** (rich

PTC UNIT - V Part-2 - File I/O

text) extension. Since text files are simple, they can be edited by any text editor like Microsoft Word, Notepad, Apple Text edit, etc.

2. **BINARY I/O stream**- It stores information as blocks or bytes in the form of **0 s** or **1 s** and it is saved with **.bin, .obj, .exe** extensions. Therefore it takes **less space**. Since it is stored in a binary number system format, it is **faster to access** and **more secure** than a text file.



What are the File Operations allowed in C?

The following is the list of file operations we can perform in the C programming language.

1. Creation of a new file (**fopen** with attributes as “w” or “w+” or “a” or “a+”)
2. Opening an existing file (**fopen**)
3. Reading from file (**fscanf** or **fgets** or **fgetc**)
4. Writing to a file (**fprintf** or **fputs** or **fputc**)
5. Moving to a specific location in a file (**fseek**, **rewind**, **ftell**)
6. Closing a file (**fclose**)
7. Removing a file (**remove**)

All the above operations are performed using the following file-handling functions available in C.

Standard Library Input Output Functions for Text & Binary Files in C

Function	Description	Syntax
File Open / Close		
fopen()	Used to create a new file or open the file if it already exists	FILE *fopen(“file_name”, “mode”);
fclose()	Used to close existing file	fclose(FILE *fp)
Formatted I/O Functions in Files		
fprintf()	Used to write formatted data in file	fprintf(FILE *stream, “formatSpecifiers”, variables)
fscanf()	Used to read formatted data from the file	fscanf(FILE *stream, “formatSpecifiers”, &variables)
Character I/O Functions in Files		
fputc()	Used to write characters in a file	fputc(char ch, FILE *stream)
fgetc()	Used to read characters from a file	fgetc(FILE *stream)
String /O Functions in Files		
fputs()	Writes a string of characters into file (not including null \0 char).	int fputs(const char *s, FILE *stream)
fgets()	Reads a string of characters from file.	char* fgets(char *s, int n, FILE *stream)
Integral File I/O Functions		
fputw()	Used to write integral value in the file	fputw(int number,File *fp)
fgetw()	Used to read integral value from the file	fgetw(File *fp)

PTC UNIT - V Part-2 - File I/O

Functions to Move within a File		
fseek()	puts the file pointer to the specified place	fseek(FILE *stream, long int offset, int whence)
ftell()	It will return the current position of the file pointer in the file	ftell(FILE *stream)
rewind()	file pointer is set to the start of the file	rewind(FILE *stream)

Modes to Open a File in C

To open a Text or Binary file we use **fopen()** with different opening modes. The most common modes used are listed below.

Mode	Description	Example
File Modes for TEXT File Streams		
r	Opens a text file in read-only mode	fopen("myfile.txt", "r")
w	Opens a text file in the write-only mode	fopen("myfile.txt", "w")
a	Opens a text file in append mode	fopen("myfile.txt", "a")
r+	Opens a text file in both reading and writing mode.	fopen("myfile.txt", "r+")
w+	Opens a text file in both reading and writing mode. It sets the cursor position to the beginning of the file if the file already exists.	fopen("myfile.txt", "w+")
a+	Opens a text file in both reading and writing mode. The reading operation is performed from the beginning and the writing operation is performed at the end of the file.	fopen("myfile.txt", "a+")
File Modes for BINARY File Streams		
rb	Opens a binary file in read mode.	fopen("myfile.txt", "rb")
wb	Opens a binary file in write mode.	fopen("myfile.txt", "wb")
ab	Opens a binary file in append mode	fopen("myfile.txt", "ab")
rb+	Opens a binary file in read and write modes	fopen("myfile.txt", "rb+")
wb+	Opens a binary file in read and write modes	fopen("myfile.txt", "wb+")
ab+	Opens a binary file in read and write modes	fopen("myfile.txt", "ab+")

How to Create or Open a File?

1. To create a new file or open an existing file, we need to create a file pointer of **FILE** type.
2. **fopen()** is a standard library function to create a new file or to open an existing file. A file can be opened in different modes such as read, write or append.

Ex:

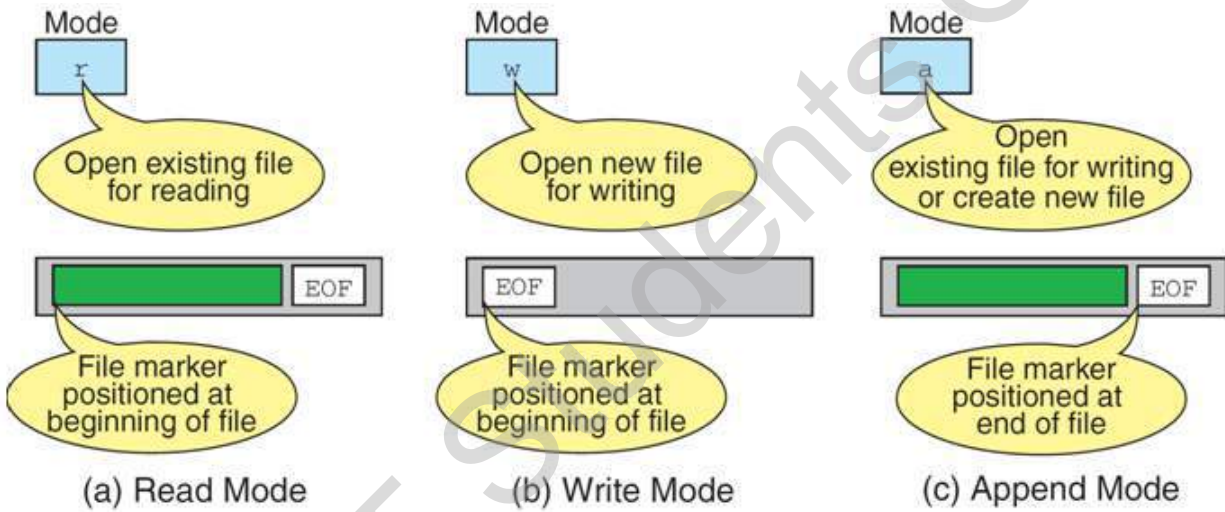
FILE *fp;

***fp = fopen("students.txt", "w");**

Explanation:

Creates a new file **students.txt** in writing mode if the file is already not available in OS.

Opens the file **students.txt** in writing mode if the file already exists in the OS.



What are Formatted Input Output Functions in Text Files?

fprintf() - This function is used to **write** formatted data **to a file** (using format specifiers).
Returns EOF in case of error writing to file.

Syntax: fprintf(FILE *stream, const char *format [, argument, ...])

Ex: `fp = fopen("file1.txt", "w");`
`fprintf(fp, "Hi Rajasekhar, This text is written by fprintf. \n");`

fscanf() - This function is used to **read** formatted data **from a file** (using format specifiers).
Returns EOF in case of error reading from file.

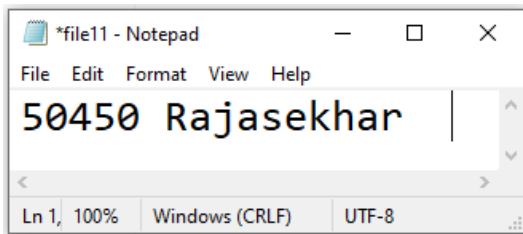
Syntax: fscanf(FILE *stream, const char *format [, argument, ...])

Ex: `fp = fopen("file1.txt", "r");`
`char buff[255];`
`fscanf(fp, "%s", buff)`

Example: Program to write to and read from a text file using fprintf() and fscanf() functions

```
#include <stdio.h>
#include <stdlib.h>
int main()
{ int num=50450;
  char name[] = "Rajasekhar";
  FILE *fp;
  fp = fopen("C:\\myfiles\\file11.txt","w");
  if(fp == NULL)
  { printf("File Error!");
    exit(1);
  }
  fprintf(fp,"%d %s",num, name); //writes num to file
  fclose(fp);
  //To read num from file
  fp = fopen("C:\\myfiles\\file11.txt","r");
  if(fp == NULL)
  { printf("Error! opening file");
    exit(1);
  }
  fscanf(fp,"%d %s", &num, name);
```

```
//screen output
printf("\n Number = %d", num);
printf("\n Name   = %s", name);
fclose(fp);
return 0;
}
```

Output Text File:**Output Screen:**

Number = 50450
Name = Rajasekhar

What are Character Input Output Functions for Text Files?

The functions **fgetc()** and **fputc()** are the character input and output functions for text files in c.

fputc() - This function is used to write/insert a character to the specified file when the file is opened in writing mode.

Syntax: fputc(char, *file_pointer)

Ex: `fp = fopen("C:\\myfiles\\file3.txt", "w");`
`fputc('A', fp);`

fgetc() - This function is used to read a character from specified file which is opened in reading mode. It reads from the current position of the cursor. After reading the character the cursor will be at next character.

Syntax: fgetc(*file_pointer)

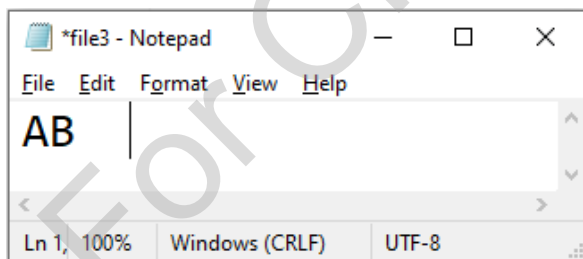
Ex: `char ch ;`
`ch = fgetc (fp) ;`

Example: Program to write and read characters to and from a text file using fputc() and fgetc()

```
#include<stdio.h>
int main( )
{
FILE *fp ;
char ch ;
fp = fopen("C:\\myfiles\\file3.txt", "w");
fputc('A', fp);
ch = 'B';
fputc(ch, fp);
fclose(fp);

fp = fopen("C:\\myfiles\\file3.txt", "r") ;
while ( 1 )
{
    ch = fgetc ( fp ) ;
    if ( ch == EOF )
        break ;
    printf("%c", ch) ;
}
fclose(fp);
return 0;
}
```

Output Text File:



Output Screen:

AB

What are String Input Output Functions for Text Files?

The functions **fgets()** and **fputs()** are the string input and output functions for text files in c.

fputs() - This function is used to insert string data into specified file that is already opened in writing mode.

Syntax: fputs("string", *file_pointer)

Ex: fp=fopen("file5.txt", "w");
fputs("Programming Through C by Dennis", fp);

fgets() - This function is used to read a string of characters from a file that is already opened. The reading starts from the current cursor position. The fgets() terminates its reading when it reaches a NULL character (\0) at the end of any string..

Syntax: fgets(stringVariableName, numberOfCharacters, *file_pointer)

Ex: printf("%s", fgets(text, 200, fp));

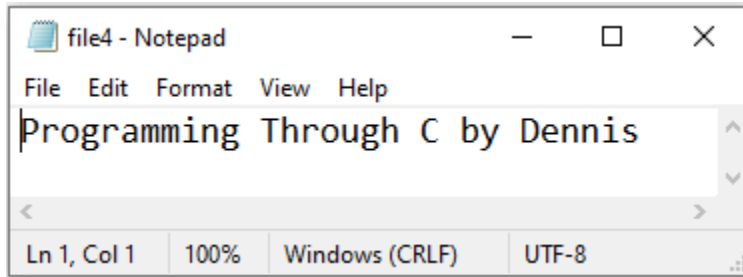
Example: Program to write and read string to and from a text file using fputs() and fgets()

```
#include<stdio.h>
int main() {
FILE *fp;
char text[300];

fp=fopen("file4.txt", "w");
fputs("Programming Through C by Dennis", fp);
fclose(fp);

fp=fopen("file4.txt", "r");
printf("%s", fgets(text, 200, fp));
fclose(fp);

return 0;
}
```


Output Text File:**Output Screen:**

Programming Through C by Dennis

What are Block Input Output Functions for Binary Files?

The functions **fwrite()** and **fread()** are used to write and read bytes of data into and from binary files.

fwrite() - This function writes bytes of data into a binary file. This functions take four parameters:

1. address of data to be written in the disk
2. size of data to be written in the disk
3. number of such type of data
4. pointer to the file where you want to write.

Syntax: fwrite(addressData, sizeData, numberOfData, pointerToFile);

fread() - This function reads bytes of data from a binary file. The function fread() also take 4 arguments similar to the fwrite() function as above.

Syntax: fread(addressData, sizeData, numberOfData, pointerToFile);

Example: Program to write into and read from a binary file using fwrite() and fread()

```
#include <stdio.h>
#include <stdlib.h>
struct marks
{
    int m1;
    int m2;
    int m3;
};
```

```

int main()
{
    struct marks std;
    FILE *fptr;
    fptr = fopen("C:\\myfiles\\file5.bin", "wb");
    if(fptr == NULL)
    {
        printf("Error! opening file");
        exit(1);
    }
    std.m1 = 100;
    std.m2 = 50;
    std.m3 = 75;
    fwrite(&std, sizeof(struct marks), 1, fptr);
    fclose(fptr);
    //To read from binary file
    fptr = fopen("C:\\myfiles\\file5.bin", "rb");
    if(fptr == NULL)
    {
        printf("Error! opening file");
        exit(1);
    }
    fread(&std, sizeof(struct marks), 1, fptr);
    printf("m1: %d\tm2: %d\tm3: %d\n", std.m1, std.m2, std.m3);
    fclose(fptr);
    return 0;
}

```

Output Binary File:

PC > Local Disk (C:) > myfiles

Name	Date modified	Type
file3	1/8/2023 10:38 PM	Text Document
file4	1/8/2023 11:25 PM	Text Document
file5.bin	1/8/2023 11:36 PM	BIN File
file5	1/8/2023 11:09 PM	Text Document
file11	1/8/2023 9:59 PM	Text Document

Output Screen:**m1: 100 m2: 50 m3: 75**

How to close a file?

The **fclose()** function is used to close an open file.

Syntax: fclose(*fp)

Ex: fclose(fp); //where fp is a file pointer

The function **fclose()** returns

- **0** on success of file close and
- **EOF** (End Of File) if any errors were detected while closing the file

Cursor Positioning Functions in Files

The following are the functions available in C to position cursor at a specific location in a file.

1. **ftell()**
2. **rewind()**
3. **fseek()**

ftell() - This function returns the current position of the cursor in the file.

Syntax: ftell(*file_pointer);

fseek() - This function is used to set the cursor position to the specific position.

Syntax: fseek(*file_pointer, int positions, int origin)

positions: indicates a number of characters to move the cursor to

origin: indicates starting place where the cursor position is measured

- 0 or SEEK_SET - from the beginning of the file
- 1 or SEEK_CUR- from the current cursor position
- 2 or SEEK_END- from the ending of the file

rewind() - This function is used reset the cursor position to the beginning of the file. But cannot be checked if it was successful. So, fseek() is preferred over rewind()

Syntax: rewind(*file_pointer)

Example: Program to show how to use fseek(), ftell() and rewind() functions

```
#include<stdio.h>
```

```
int main()
```

```
{
```

```
FILE *fp;
```

```
int position;
```

```
fp = fopen ("file6.txt", "w+"); //w+ opens file for both Read & write
```

```
fprintf(fp, "%s", "This is my file program!");
```

```
position = ftell(fp);
```

```
printf("Cursor position = %d\n", position);
```

```
rewind(fp);
```

```
position = ftell(fp);
```

```
printf("Cursor position = %d\n", position);
```

```
fseek(fp,7,0); //point at 7th position from start (0) of the file
```

```
position = ftell(fp);
```

```
printf("Cursor position = %d\n", position);
```

```
fseek(fp, -3, 2); //point at 3rd position from end (2) of the file
```

```
position = ftell(fp);
```

```
printf("Cursor position = %d\n", position);
```

```
return 0;
```

```
}
```

Output:

```
Cursor position = 24
```

```
Cursor position = 0
```

```
Cursor position = 7
```

```
Cursor position = 21
```

Program: Write a C program to copy one file into another file.

```
#include <stdio.h>
#include <stdlib.h> // To use exit()
int main()
{
FILE *fptr1, *fptr2;
char filename[100], c;
printf("Enter the filename to open for reading \n");
scanf("%s", filename);
fptr1 = fopen(filename, "r");
if (fptr1 == NULL)
{
printf("Cannot open file %s \n", filename);
exit(0);
}
printf("Enter the filename to open for writing \n");
scanf("%s", filename);
fptr2 = fopen(filename, "w");
if (fptr2 == NULL)
{
printf("Cannot open file %s \n", filename);
exit(0);
}
c = fgetc(fptr1);
while (c != EOF)
{
fputc(c, fptr2);
c = fgetc(fptr1);
}
printf("\nContents copied to %s", filename);
fclose(fptr1);
fclose(fptr2);
return 0;
}
```

Output:

Enter the filename to open for reading

file1.txt

Enter the filename to open for writing

file2.txt

Contents copied to file2.txt

Program: Write a C program to remove a file from the disk.

```
#include <stdio.h>
int main()
{
int status;
char fname[20];
printf("\n\n Remove a file from the disk :\n");
printf(" \n");
printf(" Input the name of file to delete : ");
scanf("%s",fname);
status=remove(fname);
if(status==0)
{
printf("\nFile %s has been removed.",fname);
}
else
{
printf("\nFile %s hasn't been found or does not exist.",fname);
}
return 0;
}
```

Output1:

Remove a file from the disk :
 Input the name of file to delete : **file3.txt**
 File file3.txt has been removed.

Output2:

Remove a file from the disk :
 Input the name of file to delete : **file3.txt**
 File file3.txt hasn't been found or does not exist.